# Løsningsforslag TDT4127 Høst 2018

## Oppgave 1: Python-teori

1) Hva er forskjellen på lister og tupler i Python?
**Korrekt svar: Lister er muterbare, mens tupler er ikke-muterbare**

2) Mengder (eng. set) i Python har...
**Korrekt svar: ingen dupliserte elementer, og ikke noen fast rekkefølge på elementene**

3) 3.25e19 i Python betyr det samme som
**Korrekt svar: 3.25*10**19**

4) For løsning av andregradslikninger brukes typisk formelen

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

For hvilken kombinasjon av variabelverdier vil denne formelen gi svært dårlig presisjon hvis vi oversetter den direkte til Python, f.eks. x1 = (-b - math.sqrt(b**2 - 4*a*c))/(2*a) og x2 = (-b + math.sqrt(b**2 - 4*a*c))/(2*a) ?
**Korrekt svar: a = 1, b = $10^8$, c = 1**
(grunnen til dette er at her blir b**2 svært dominerende i kvadratrotuttrykket, slik at vi subtraherer to nesten like tall, dermed mister vi presisjon)

5) Tilordningen X = { } i Python oppretter X som
**Korrekt svar: en tom dictionary**

## Oppgave 2: Numerikk-teori

1) Hva kjennetegner kompositttmetoder for numerisk integrasjon?
**Korrekt svar: Kompositttmetoder baserer seg på å splitte integrasjonsintervallet i flere subintervaller**

2) Rangér metodene fra mest til minst restriktive konvergensbetingelser.
**Korrekt svar: Newton > sekant > biseksjon**

3) Hva er usant om algoritmer for å finne nullpunkt til ikke-lineære funksjoner?
**Korrekt svar: Biseksjonsmetoden baserer seg på å følge tangentlinjene til funksjonen**

4) Hva er hensikten med partiell pivotering i Gauss-eliminasjon?
**Korrekt svar: Å redusere regnefeil som følge av flyttallspresisjon**

5) Heuns metode er et eksempel på...
**Korrekt svar: En to-stegs metode**

6) Adaptiv Simpsons metode er...
**Korrekt svar: Et eksempel på en rekursiv algoritme**

7) Hva er usant om flyttall?
**Korrekt svar: Det er uproblematisk å sjekke likhet mellom to flyttall**

8) Hvilken av disse algoritmene brukes for å beregne bestemte integraler?
**Korrekt svar: Trapesmetoden**

9) Hva er en rimelig kontrollstruktur å bruke når man skal implementere Newtons metode?
**Korrekt svar: En while-løkke**

10) Hva kan man ikke bruke feilestimater for integrasjonsmetoder til?
**Korrekt svar: Gi eksakte svar på integralene**

## Oppgave 3: Adaptiv Simpson

Oppgave: Vil koden for adaptiv Simpson avsluttes dersom den kjøres? Svar ja/nei.

**Korrekt svar: Nei. Koden vil alltid gjøre nye rekursive kall før den kommer til en return-linje og dermed aldri avslutte - den fortsetter bare å gjøre nye kall som gjør nye kall som gjør nye kall ad infinitum.**

## Oppgave 4: Gauss-eliminasjon

Oppgave: For hver funksjon under, finn kodefeilen.

```
def add(row1, row2, num):
     for i in range(0,len(row2)):
          row2[i] = row1[i] + num*row2[i]
          # Feil: Skulle vært row2[i] + num*row1[i]

def swap(row1,row2):
     for i in range(0,len(row2)):
          temp = row2[i]
```

```
            row1[i] = temp
            row2[i] = row1[i]
            # Feil: De to siste linjene skal byttes, ellers
            ender man opp med innholdet fra row2 i begge
            vektorene.

def get_max_row(A,row,col):
      currentMax = abs(A[row][col])
      currentMaxIndex = row
      for i in range(row+1,len(A)):
            if abs(A[i][col]) > currentMax:
                  currentMax = abs(A[col][i])
                  # Feil: Skulle vært abs(A[i][col]), col angir
                  kolonneindeksen, ikke radindeks.
                  currentMaxIndex = i
      return currentMaxIndex
```

**Svar:** Se kommentarer i koden over.

## Oppgave 5: Filer, feilfinning

1) Hva er grunnen til SyntaxError her?
   **Korrekt svar: Det mangler et komma i print-setningen**

2) Hva er det som forårsaker TypeError her?
   **Korrekt svar: Det som står inni num_list, er ikke blitt konvertert til heltall, så det står fortsatt som strenger: ['1', '2', '0', '0','3']**
   Grunnen til dette er at når man itererer gjennom en liste med en setning av typen
   `for element in liste:`
   så vil dette bare funke for å se elementene, men ikke for å endre dem. For å endre måtte vi ha aksessert elementene via indeks.

3) Hva er det som gjør at «Jo» får ValueError her?
   **Korrekt svar: Setningen f.close() utføres før vi har lest hele fila**
   (Grunnen er at den står på innrykk og dermed kommer inn i løkka, skulle vært en marg lenger ute så den kom etter løkka)

4) Hva er det som gjør at summen bare blir 9?
   **Korrekt svar: Det brukes feil operator i linja total_sum =+ sum(num_list), skulle vært +=**

## Oppgave 6 Lister, mengder, in-operator

*(Nedenfor vises skjermbilde fra Inspera av oppgaven, med riktige ruter fylt inn. På hver enkelt sin eksamen vil radene ha kommet i tilfeldig rekkefølge, så avkrysningsmønsteret kan ha vært et annet)*

Vi har definert tre globale variable:

- A = {0,1,2,3}
- B = {3,4,5}
- C = set(range(8))

Vi har dessuten funksjonen **count_list(lst)** gitt som følger:

```
def count_list(lst):
    result = [0]*5
    for s in lst:
        for i in range(1,6):
            if i in s:
                result[i-1] += 1
    return result
```

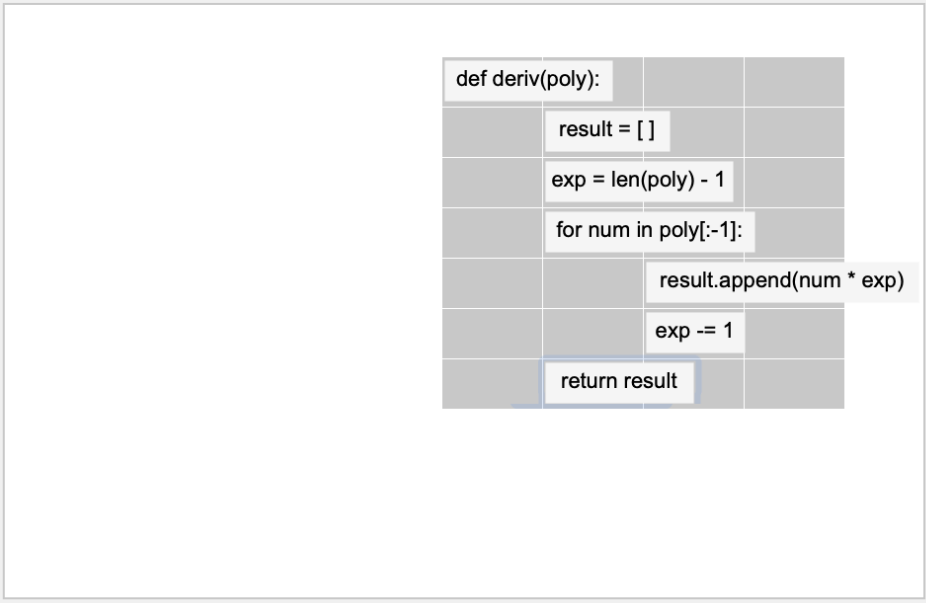
**Hver rad-tittel i tabellen er et mulig argument til funksjonen count_list(). Hver kolonneoverskrift er en mulig returverdi. Marker hvilket argument som fører til hvilken returverdi.**

| | [1,1,1,0,0] | [1,1,1,1,1] | [0,0,0,0,0] | [2,2,3,2,2] | [6,6,6,6,6] |
|---|---|---|---|---|---|
| [A] | ◉ | ○ | ○ | ○ | ○ |
| [A.union(B)] | ○ | ◉ | ○ | ○ | ○ |
| [A.difference(C)] | ○ | ○ | ◉ | ○ | ○ |
| [A,B,C] | ○ | ○ | ○ | ◉ | ○ |
| [C]*3+[C]*3 | ○ | ○ | ○ | ○ | ◉ |

## 7) Symbolsk derivasjon

(kodelinje 2 og 3 i omvendt rekkefølge ville også gi riktig svar)

**NB! For å velge riktig innrykk må du passe på at midten av kodelinjen er over riktig boks når den slippes. Kodelinjen vil da automatisk settes på plass.**




## 8) Polynomledd

**Oppgave:** *Velg riktig i de seks feltene nedenfor slik at funksjonen virker korrekt.*

```
def poly_part(c, n):
    result = ""
    if [c != 0:]
        if n == 0:
            result = [str(c)]
        else:
            if c == -1:
                [result = "-x"]
            elif c == 1:
                result = "x"
            elif [abs(c) > 1:]:
                result = [str(c) + "*x"]
            if n > 1:
                [result += "**" + str(n)]
    return result
```

## 9) Lage streng...

**Oppgave:** *Velg kodelinjer i de fem feltene slik at funksjonen fungerer.*

```
def poly_str(lst):
    result = ""
    for p in range(len(lst)):
        num = lst[-1-p]
        if num != 0:
            part = poly_part(num, p)
            if result == "" or result[0] == "-":
                result = part + result
            else:
                result = part + "+" + result
    return result
```

Koden funker ved å gå i løkke og legge til ett og ett ledd i polynomstrengen. Starter derfor med å opprette en tom streng i første nedtrekkboks. I andre nedtrekkboks blir korrekt indeks -1-p fordi løkka må hente ut elementene baklengs fra lista lst hvis koden skal virke. At det må skje baklengs kan man utlede fra følgende: (i) når uttrykket bygges opp i nest siste kodelinje legges den neste delen av strengen (part) til foran det som allerede er samlet opp i result, og (ii) p, som er indeks i for-løkka, brukes som potens for leddene i kallet poly_part(num, p); p vil dermed starte på 0 som tilsier det bakerste leddet i lst. I tredje boks må vi ha num != 0, dette blir betingelsen for å legge til et ledd (ellers er det bare tallet null, som ikke skal med). Nedtrekksboksen for if-betingelsen kan være litt tricky, poenget her er at else-alternativet innebærer at vi legger til et plusstegn i strengen. If-betingelsen må da uttrykke hva som er tilfelle når vi ikke skal legge til et plusstegn. Dette vil være enten hvis det fremste tegnet i strengen er minus, eller hvis vi foreløpig har en tom streng slik at leddet vi nå kommer med blir det første hittil (så det ikke er to ledd å sette plusstegn mellom).

## 10) Finne tall

```
def find_expo(expr):
    if "**" in expr:
        return int(expr.split("**")[1])
    elif "x" in expr:
        return 1
    else:
        return 0

def find_const(expr):
    if "*x" in expr:
        return int(expr.split("*x")[0])
    elif "-x" in expr:
        return -1
    elif "x" in expr:
        return 1
    else:
        return int(expr)
```

For find_expo() vil man i det generelle tilfellet ha et potenstegn, og det enkleste er da å splitte på potenstegnet, hvorpå potensen vil være ledd [1] i den resulterende lista mens alt foran potensen vil være ledd [0]. Å splitte på enkelt gangetegn er også ok, men må da bruke indeks [-1] for siste element siden det vil lages en liste med mer enn to element. Må også huske på spesialtilfeller hvor det ikke fins noe potenstegn. Hvis uttrykket inneholder x vil potensen da være 1, hvis ikke er potensen 0.

For find_const() er det likeledes noen spesialtilfeller man må ta hensyn til for å få full score. Hvis uttrykket innholder en multiplikasjon med x, kan man splitte på *x og konstanten vil da stå fremst i den resulterende lista. Ellers har vi spesialtilfeller med at det bare står x eller –x eller kun en konstant.

## 11) Liste

```
def poly_list(expr):
    result = [ ]
    start = 0
    for i in range(1, len(expr)):
        if expr[i] == '+' or expr[i] == '-':
            result.append(expr[start:i])
            if expr[i] == '+':
                start = i + 1
            else:
                start = i
    result.append(expr[start:len(expr)])
    return result
```

Poenget med `start = i + 1` vs `start = i` er at minustegnet må tas med i den resulterende lista, mens plusstegnet ikke skal med (og derfor +1, dvs. start for neste delstreng settes til posisjon etter +tegnet)

## 12) Fra uttrykk til liste

```
def expr_2_list(expr):
    ex_list = poly_list(expr)
    result = [ ]
    for i in range(len(ex_list)-1, -1, -1):
        # itererer baklengs, starter på laveste potens
        expo = find_expo(ex_list[i])
        const = find_const(ex_list[i])
        for i in range(len(result), expo):
            # legger inn 0'er for potensledd som mangler
            result = [0] + result
        # setter inn konstant for nytt ledd fremst i lista
        result = [const] + result
    return result
```

Det er også fullt mulig å løse oppgaven ved å iterere forlengs gjennom lista, utfordringen med å få satt inn ekstra nuller blir om trent den samme men kanskje litt mer tricky da man må sammenligne to forrige og denne konstant, mens baklengsmetoden kan se på lengden av resultatvektoren til en hver tid.

## 13) Newton

Oppgave: Dra og slipp linjene så de havner i riktig rekkefølge og med riktig innrykk, slik at funksjonen implementerer Newtons metode.

**Korrekt svar:**

```
def Newton_method(f,df,x,epsilon):
     while(abs(f(x)) > epsilon):
          if abs(df(x)) > 1E-10:
               x = x - f(x)/df(x)
          else:
               print("Error!"); return
     return x
```

**NB: Noen kandidater fant en fungerende omstokking av koden, disse har fått full uttelling på denne oppgaven.**

## 14) Eksplisitt Euler

Oppgave: Implementér funksjonen explicit_Euler_list(F,x,N,T).

**Korrekt svar:**

```
def explicit_Euler_list(F,x,N,T):
    h = T/N
    x_list = [x]                # Legger inn startverdi i listen
    for k in range(N):
        t_k = k*h               # Tidspunkt for den deriverte
        x = x  + h*F(x,t_k)     # Gjennomfører steget
        x_list.append(x)        # Legger inn ny x-verdi i listen
    return x_list               # Returnerer listen
```

## 15) Integrasjon

Oppgave: *Velg riktige alternativer fra menyene slik at* **comp_simpson_method(x_list,a,b)** *implementerer Simpsons metode der* **x_list** *er en liste [*x0,x1,...,xN*] som inneholder punktene som skal brukes i formelen for Simpsons metode, og* **a** *og* **b** *er venstre og høyre endepunkt i integralet.*

**Korrekt svar:**

```
def comp_simpson_method(x_list,a,b):
    N = len(x_list)-1 # Listen er [x_0,…,x_N], altså N+1 tall
    h = (b-a)/float(N)
    total_sum = x_list[1] # Første ledd i summen
    for j in range(1,N):
        if j % 2 is 0:              # Partallsindeks
            total_sum += 2*x_list[j]
        else:
            total_sum += 4*x_list[j] # Oddetallsindeks
    total_sum += x_list[N]
    total_sum = h/3*total_sum       # Multipliserer summen
    return total_sum
```

## 16) Testfunksjon

**Oppgave:** *Implementér en funksjon* **test_functions()** *uten input som tester funksjonene fra de foregående oppgavene. Du kan anta at g(x) og G(x,t) allerede er implementert.*

**Korrekt svar: Koden under er et eksempel på hvordan denne oppgaven kan løses, men er ikke nødvendigvis den eneste korrekte måten. Det var også et par uregelmessigheter i oppgaveteksten:**

**random-biblioteket var ikke inkludert i vedlegget som kom med på eksamen, dette har blitt tatt høyde for i rettingen. Det samme gjelder bruk av dg i newton-metoden - det var ikke informert om at derivert var implementert, og det er tatt høyde for i retting. I tillegg kunne variabelnavnet «x_list» i comp_simpson_method være forvirrende da det samme navnet var brukt i explicit_Euler_list, vi har tatt høyde for dette i rettingen.**

```
import math
import random
def test_function():
     for i in range(20):
          x_0 = random.uniform(-5, 5)
          x = Newton_method(g,dg,x_0,epsilon = 0.001)
          print('Newtons metode med startpunkt p =',x_0,
          'konvergerte med endelig funksjonsverdi g(x) =',
          g(x))
     for j in range(1,6):
          x_list = explicit_Euler_list(G,x =0,N = 10**j,T = 1)
          error = abs(x_list[-1]- (-math.exp(-1)))
          print('Feilen ved tidspunkt T = 1:',error)
     x_list = [g(i/20) for i in range(21)]
     integral = comp_simpson_method(x_list,0,1)
     error = abs(integral-(math.exp(1)-3))
     print('Feilen i Simpsons metode er:',error)
```

## 17) Presentasjon av resultat

**Oppgave:** *Implementér funksjonen* **function_info(f,a,b)** *som oppfyller spesifikasjonene i oppgaven.*

**Korrekt svar: Koden under er et eksempel på hvordan denne oppgaven kan løses, men er ikke nødvendigvis den eneste korrekte måten – det er for eksempel flere måter å komme fram til samme plot.**

**Også her har vi tatt høyde for at variabelnavnet x_list i inputene til comp_simpson_method kan være forvirrende.**

```
import matplotlib.pyplot as plt

def function_info(f,df,a,b,T):
    z = Newton_method(f, df, x = (a+b)/2, epsilon = 0.001)
    print('Nullpunktet er nært x =',z)
    x_list = [f(i/100) for i in range(101)]
    totsum = comp_simpson_method(x_list, a, b)
    print('Integralet har tilnærmet verdi:',totsum)
    x = explicit_Euler_list(f, 1, 100, T)
    h = T/100
    t = [h*i for i in range(101)]
    plt.plot(t,x,'g-')
    plt.xlabel('t')
    plt.ylabel('x(t)')
    plt.title('Steglengde h = '+str(h))
    plt.show()
```

## 18) Enhetsmatrise

Flere mulige måter å løse denne på. Enten starte med ei tom liste og så legge til ett og ett element, 0 eller 1 alt etter som:

```
def unit_matrix(n):
    matrix = []
    for i in range(n):
        matrix.append([])
        for j in range(n):
            if i == j:
                matrix[i].append(1)
            else:
                matrix[i].append(0)
    return matrix
```

Eller lage en matrise med bare nuller først og så gå gjennom denne og putte 1'ere akkurat i diagonalen. F.eks.

```
def unit_matrix(n):
    matrix = []
    for i in range(n): # matrix of n x n zeros
        matrix.append([0]*n)
    for i in range(n): # sets diagonal to 1
        matrix[i][i] = 1
    return matrix
```

Den innledende nullmatrisa kan også lages med en list comprehension,
`matrix = [[0 for i in range(n)] for j in range(n)]`
Merk derimot at det ikke vil fungere å lage nullmatrisen ved `matrix = [[0]*n]*n`
Da dette gjør at alle de indre listene egentlig viser til samme liste, så man får et sluttresultat der alle tall i matrisa er 1 (eller evt. alle er 0), ikke en enhetsmatrise.

Hvis man først skal bruke list comprehension, kan man evt. gjøre det eksta kompakt og elegant ved å lage enhetsmatrisa med en enkelt programsetning:

```
def unit_matrix3(n):
    return [[int(i==j) for i in range(n)] for j in range(n)]
```

## 19) Sjekke enhetsmatrise

```
def is_unit_matrix(A):
    rows = len( A )
    for i in range(rows):
        if A[ i ] != [0] * i + [1] + [ 0 ] * (rows -i-1 ):
            return False
    return True
```

Figuren viser riktig utfylt oppgave. I ruta der det står -i-1 ville også tre andre svar være riktige: -1-i, -(i+1), -(1+i) siden alle disse også gir samme resultat. Ideen bak funksjonen er å gå i løkke rad for rad gjennom matrisa og sjekke at hver rad stemmer med tilsvarende rad i en enhetsmatrise. I enhetsmatrisa skal 1'eren alltid stå på indeks i mens øvrige tall er 0, dvs. vi må først ha i 0'ere, så 1'eren, og så de resterende 0'ere, som blir totalt antall element som en rad skal ha, fratrukket de i elementene vi hadde før 1'eren, og 1'eren selv, altså rows-i-1.

En del har skrevet A[0] i første boks hvor svaret skal være A. A[0] vil funke HVIS A er en enhetsmatrise, men er generelt feil fordi funksjonen feilaktig vil kunne svare True på en del matriser som ikke er enhetsmatriser. F.eks. [[1,0,0],[0,1,0],[0,0,1],[1,2,3]] vil feilaktig evaluere til True hvis vi bruker A[0] fordi løkka da bare sjekker de første tre radene og ignorerer at det fins en fjerde rad som ikke passer inn. Med A ville denne derimot korrekt evaluere til False.

# 20) Glisne matriser

## Del 1, konvertering

Funksjon for å konvertere en matrise representert på vanlig måte til den mer kompakte representasjonen:

```
def comp_mat(M):
    rows = len(M)
    cols = len(M[0])
    result = [[rows,cols)], [], [], [] ]
    for i in range(rows):
        for j in range(cols):
            if M[i][j] != 0:
                result[1].append(i)
                result[2].append(j)
                result[3].append(M[i][j])
    return result
```

Her er ideen å starte med en liste som inneholder dimensjonene, samt tre tomme lister for å inneholde hhv. radindeks, kolonneindeks og verdier. Deretter går vi i dobbel løkke gjennom matrisa og legger inn disse tallene hver gang vi finner et element som ikke er null.

## Del 2, Addering av to kompakte matriser

Denne oppgaven var ment å være ekstra vanskelig, så bare de aller flinkeste ville være i nærheten av å få den til. Mens det er ganske enkelt (en rett fram dobbel løkke) å addere to matriser representert ved vanlige 2D lister, er addisjon av to matriser på den kompakte formen gitt i denne deloppgaven langt mer krøkkete (dvs., koden blir mer kompleks, men mer tidseffektiv, siden vi slipper unna med å bare se på elementer som ikke er null, og ikke minst mer plasseffektiv siden vi slipper å lagre null-elementene). Å konvertere matrisene tilbake til normal representasjon, utføre addisjonen der, og så konvertere tilbake til kompakt representasjon, vil derfor ikke være en spesielt god løsning – siden da mister man hele denne besparelsen av plass og tid.

Vi lager først en liten hjelpefunksjon `pos()` som skal fortelle indeksposisjonen (rad, kolonne) til element nr i for matrise M (A eller B). Denne hjelpefunksjonen er ikke strengt tatt nødvendig, men sparer oss for en del indekser slik at betingelser i if- og while-setninger i resten av koden blir enklere å lese. I begge løsningsvariantene nedenfor antas at element i A og B står i sortert rekkefølge fra lave til høye indeksverdier (først sortert på rad, så kolonne), og resultatet leveres også tilsvarende sortert.

```
def pos(M, i):
    return (M[1][i], M[2][i])

# VARIANT 1 av løsning, starte med tomme lister,
# legge til ett og ett element med append()

def add_comp2(A, B):
    if A[0] == B[0]:
        C = [list(A[0]), [], [], []]
        a = 0
        b = 0
        while a < len(A[1]) or b < len(B[1]):
            if b >= len(B[1]) or pos(A,a) < pos(B,b):
                new_element = [A[1][a],A[2][a],A[3][a]]
                a += 1
            elif a >= len(A[1]) or pos(B,b) < pos(A,a):
                new_element = [B[1][b],B[2][b],B[3][b]]
                b += 1
            else: # equal position, add values
                value = A[3][a] + B[3][b]
                if value != 0:
                    new_element = [A[1][a],A[2][a],value]
                else: # sum was zero, no new element
                    new_element = []
                a += 1
                b += 1
            for i in range(len(new_element)):
                C[i+1].append(new_element[i])
        return C
    else:
        return False
```

I denne varianten starter vi med tomme lister (bortsett fra å kopiere dimensjonene), deretter fletter vi oss gjennom A og B og tar i hver runde av løkka elementet med minste indekser. Den ytre while-løkka tester at vi fortsatt har element igjen i minst en av matrisene. If-testen sjekker så hva som har lavest indeks, elementet i A, elementet i B, eller begge på samme indeks. Det laveste velges som new_element som er det som skal inn i C nå… hvis begge var likt, summeres verdien – merk dog at hvis denne summen blir null, skal elementet ikke settes inn i det hele tatt. Tilsvarende økes indeks a, b eller begge for å gå til neste element vi skal se på. For-løkka nederst i while-løkka setter new_element inn i C.

```
#VARIANT 2 av løsning: kopierer først resultatmatrisa C fra A,
#Går deretter i løkke gjennom B og setter inn elementene på
#riktig plass i C.

def add_comp(A, B):
    if A[0] == B[0]:
        C = [list(A[0]), list(A[1]), list(A[2]), list(A[3])]
        i = 0
        for b in range(len((B[1]))):
            while i < len(C[1]) and pos(C,i) < pos(B,b):
                i += 1 # element in B comes later
            if pos(C,i) == pos(B,b): # same pos, add value
                C[3][i] += B[3][b]
                if C[3][i] == 0: #sum was zero, remove element
                    for j in range(1,4):
                        del C[j][i]
                    i -= 1
            else: #other position, insert element from B
                for j in range(1,4):
                    C[j].insert(i, B[j][b])
            i += 1
        return C
    else:
        return False
```

Tilordningen C = [list…] gjøres for å ta en kopi av C, hvis vi bare gjorde C = A ville C og A være samme liste, så endringer av C også ville endre A, som kanskje ikke var meningen. Går så i for-løkke gjennom elementene i B. while-løkka hopper forbi elementer i C som har lavere indeks, finner dermed riktig posisjon hvor elementet fra B skal inn. Når riktig posisjon er funnet, sjekkes om B og C har samme posisjon, da summeres verdiene, hvis denne summen ble null, må elementet deretter fjernes. Hvis derimot posisjonen ikke er lik, skal elementet fra B settes inn i C med insert(). Indeksen i holder rede på hvilken posisjon som er neste for innsetting i C, og når vi fjerner et element med del må i reduseres med 1 (for å kompensere for at vi nederst i for-løkka øker den med 1; da fjerning vil gjøre at neste element havner på indeks i direkte).

Av de to alternative løsningene har VARIANT 2 litt kortere kode, men den er egentlig en dårligere løsning. Dette er fordi VARIANT 2 bruker `insert()` for å sette inn element i C, og `del` for å fjerne element hvis summen ble null. Begge disse operasjonene gjør endringer

midt i lista, som er forholdsvis tidkrevende under panseret pga behov for å flytte påfølgende elementer i lista. (Mer presist har disse operasjonene en algoritmekompleksitet på O(n) hvor n er antall elementer i lista). Særlig hvis det er snakk om store matriser er VARIANT 1 mer effektiv med tanke på kjøretid fordi den kun benytter `append()` som legger til element bakerst i listene, som dermed er en mye «billigere» operasjon å utføre (Mer presist er append bare O(1) mhp algoritmekompleksitet). Gitt oppgavens vanskegrad ville imidlertid begge disse løsningene – eller noe nærliggende – gi full score.